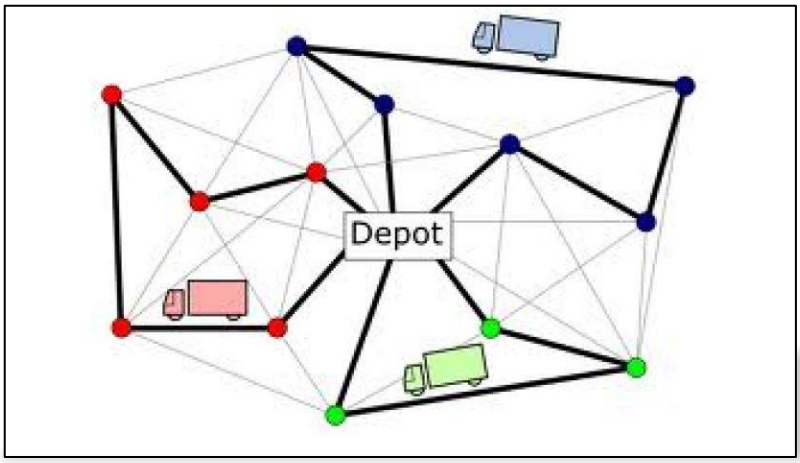
CLERMONT AUVERGNE INP Isima

Enseignant : Philippe Lacomme



TSP, VRP et Recherche Opérationnelle

TP3 OUTILS D'AIDE A LA DECISION

Anaël Courjaud | S1 ZZ2 F2 | 18/01/2023

Table des matières

I. Introduction

Durant la quasi-totalité de la phase de développement de mon code, j'ai pris l'instance VRP_DLP_01.txt (./Code source Linux/instances/en-dessous de 100 villes/) pour effectuer mes tests. J'ai souvent réajusté les ambitions de mon code en fonction des durées nécessaires à ces tests. Comme cette instance a un nombre de sommets pas très élevé (91 contre 255 pour la pire instance du lot), sachant que la complexité (donc le temps d'exécution) ne croît pas de façon linéaire en fonction de la taille de l'instance, mais plutôt exponentiellement, il se peut donc que mon programme ne soit pas vraiment adapté en termes de complexité et de durée d'exécution à des énormes instances. Heureusement, mes fonctions les plus gourmandes sont entièrement paramétrables dans le fichier general.h et il existe quelques petites astuces pour combattre ces problèmes de gourmandise (voir README de mon code source Linux, ou alors les commentaires dans general.h).

II. Enjeux et objectifs du TP

A. TSP ? VRP ?

Tous deux faisant partie de la classe des problèmes NP-complets, le TSP (Traveling Salesman Problem - Problème du Voyageur de Commerce) et le VRP (Vehicle Routing Problem – Problème de Routage de Véhicules) sont connus pour être impossibles à résoudre de façon exacte en un temps raisonnable, à partir d'une certaine taille d'instance. En effet, vu l'explosion combinatoire du nombre de solutions différentes envisageables en fonction de la taille de l'instance, on se retrouve très vite avec un temps de calcul presque infini (voir Figure 1) ! Il va donc falloir ruser et avoir des stratégies calculatoires très réfléchies avant de pouvoir espérer obtenir des solutions de bonne qualité.

Figure 1 : Estimation du nombre de solutions différentes pour un TSP en fonction de la taille de son instance, n incluant le dépôt (source Wikipédia)

Explosion combinatoire

Ce problème est plus compliqué qu'il n'y paraît ; on ne connaît pas de méthode de résolution permettant d'obtenir des solutions exactes en un temps raisonnable pour de grandes instances (grand nombre de villes) du problème. Pour ces grandes instances, on devra donc souvent se contenter de solutions *approchées*, car on se retrouve face à une explosion combinatoire.

Pour un ensemble de n points, il existe au total $n!$ chemins possibles (factorielle de n). Le point de départ ne changeant pas la longueur du chemin, on peut choisir celui-ci de façon arbitraire, on a ainsi $(n-1)!$ chemins différents. Enfin, chaque chemin pouvant être parcouru dans deux sens et les deux possibilités ayant la même longueur, on peut diviser ce nombre par deux. Par exemple, si on nomme les points, a, b, c, d, les chemins abcd et dcba, cdab et badc, adcb et bcda, cbad et dabc ont tous la même longueur ; seul le point de départ et le sens de parcours changent. On a donc $\frac{(n-1)!}{2}$ chemins candidats à considérer .

Par exemple, pour 71 villes, le nombre de chemins candidats est supérieur à 5×10^{80} qui est environ le nombre d'atomes dans l'univers connu .

Nombre de chemins candidats en fonction du nombre de villes

Nombre de villes n	Nombre de chemins candidats $\frac{1}{2}(n-1)!$
3	1
4	3
5	12
6	60
7	360
8	2 520
9	20 160
10	181 440
15	43 589 145 600
20	$6{,}082 \times 10^{16}$
71	$5{,}989 \times 10^{99}$

Pour résumer grossièrement, le TSP modélise le problème que peut avoir un voyageur de commerce lors de la planification de son trajet avant son départ. Il doit trouver l'ordonnancement optimal de ses étapes (pour la suite du rapport, étape, ville ou encore sommet signifieront la même chose) tout en réduisant au maximum le coût de son voyage (ici, la dépense est proportionnelle au nombre de kilomètres parcourus). Un tel ordonnancement, qu'il soit optimal ou non, est appelé « Tour Géant » puisqu'il passe par toutes les étapes avant de retourner à son point de départ.

Le VRP quant à lui, nous met dans la peau d'une société de livraison de colis. On connaît le volume demandé à chaque étape ainsi que le volume maximum que peut embarquer un véhicule. Le but est donc de trouver un ensemble de « petits » Tours Géants (des tournées) permettant de livrer tous les colis demandés à chacune des étapes avec un coût minimal (encore une fois en fonction de la distance parcourue). Un véhicule pourra donc effectuer les différentes tournées sans être en surcharge, tout en retournant au dépôt entre chacune d'elles pour charger la cargaison destinée à la prochaine tournée.

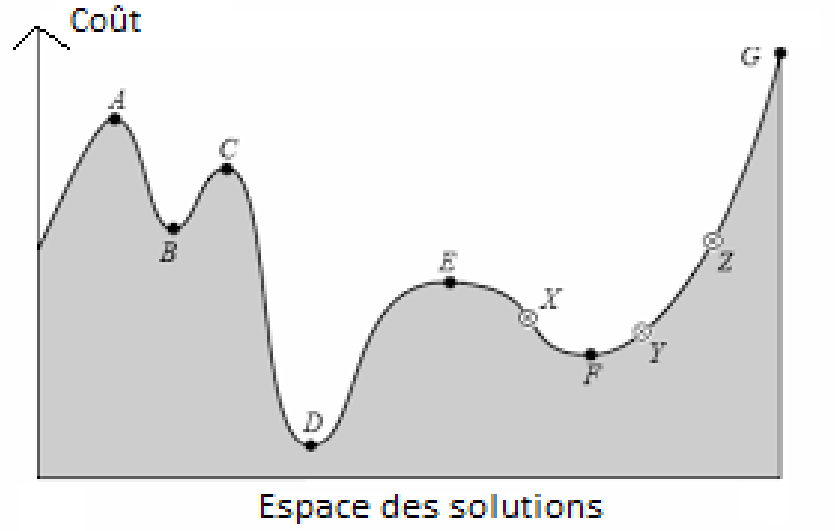
Il existe évidemment plein d'autres contraintes que l'on peut ajouter aux modèles précédent pour se rapprocher de la réalité, mais cela aura presque automatiquement pour effet d'encore plus complexifier le problème en augmentant le nombre de solutions envisageables et donc d'allonger les temps de calculs. Pour le VRP, on peut par exemple introduire de nouveaux véhicules avec des caractéristiques différentes comme par exemple plus de capacité de chargement mais aussi plus de consommation de carburant, ce qui aura pour effet d'augmenter le coût en fonction de la distance parcourue etc…

Enfin bref, les possibilités sont infinies et il est très important de bien définir les contraintes du problème en amont.

B. ESPACE DES SOLUTIONS, MINIMAS LOCAUX/GLOBAUX ET EXPLORATION DU VOISINAGE

Un autre gros problème en recherche opérationnelle réside dans la « cartographie » de l'espace des solutions et dans l'impossibilité de déterminer si un minimum local est également global. A l'inverse, il est parfois très facile de trouver le maximum global (la pire solution possible) mais ce n'est malheureusement pas très productif ☺

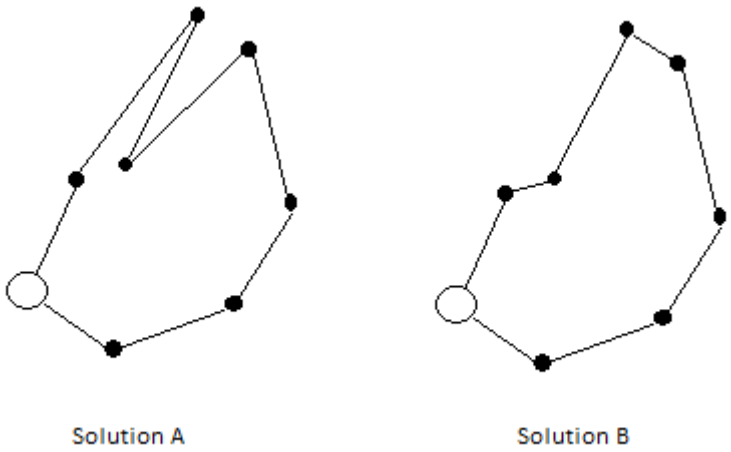
Figure 2 : Vulgarisation de l'Espace des solutions et des coûts associés



La Figure 2 montre différentes solutions avec leur coût associé. On peut déterminer graphiquement que G est le maximum global ; D est le minimum global ; B et F sont des minimums locaux ; F, Y et X sont voisins etc… Mais que signifie concrètement le fait que deux solutions soient voisines ? Deux solutions d'une même instance sont voisines si elles se « ressemblent » beaucoup.

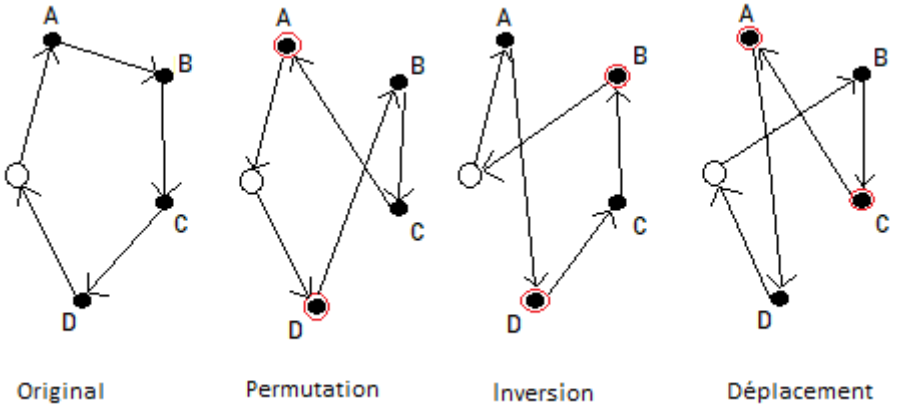
Les solutions A et B de la Figure 3 sont très semblables : il suffit de permuter deux étapes dans l'ordre de passage pour passer d'une solution à l'autre. Elles n'ont en revanche pas du tout le même coût car on voit bien que la solution B est bien plus optimisée que la solution A ! Le caractère voisin de deux solutions n'est donc pas vraiment lié à leurs coûts mais plutôt au nombre de perturbations qu'il faut leur faire subir afin de passer de l'une à l'autre.

Figure 3 : Instance de TSP à 7 sommets, les deux solutions proposées sont voisines par permutation



On peut donc « explorer » aisément le voisinage d'une solution en générant des voisins par des simples perturbations. La Figure 4 explicite les différents types de perturbations appliquées par mon code lors de ces phases d'exploration.

Figure 4 : Différentes perturbations utilisées par mon code lors des phases d'exploration



C. ALGORITHME DE DESCENTE ET RECHERCHE LOCALE

Ce principe d'exploration du voisinage d'une solution par petites perturbations va nous permettre de mettre en œuvre ce qu'on appelle un algorithme de descente, dont on va illustrer le fonctionnement avec l'exemple suivant :

L'algorithme, appliqué à l'instance dont l'espace des solutions est représenté par la Figure 2, aura ici pour objectif de trouver le minimum global (graphiquement la solution D) en partant de la solution G. Admettons qu'en explorant le voisinage de G de façon aléatoire, on finisse par tomber sur la solution Z. Son coût étant moindre on abandonne alors l'exploration du voisinage de G pour se concentrer sur l'exploration du voisinage de Z. En réitérant ce processus on va finir par tomber sur la solution F. Lors de l'exploration de son voisinage, on se rend alors compte qu'aucun des voisins de F n'a un coût moindre. F est donc un minimum local, au grand dam de notre algorithme de descente, qui se retrouve « bloqué » dans ce minimum local et qui n'arrivera jamais à atteindre la solution D. En revanche, en partant de la solution E, l'algorithme avait de bonnes chances de finir par trouver la solution D, à condition de ne pas partir dans la « mauvaise direction », ce qui est malheureusement assez imprévisible. Cette façon d'utiliser un algorithme de descente s'appelle une Recherche Locale et nous allons utiliser ce principe pour les fonctions *RL_nomDeLaFonction()* de mon code source.

D. REFLEXION SUR LES STRATEGIES A ADOPTER

En reconsidérant la Figure 2 à la lumière de ce nouvel outil qu'est la recherche locale, on se rend rapidement compte que pour réussir à trouver le minimum global D (« tomber dans la crevasse la plus profonde »), il faudra forcément avoir lancé une recherche locale sur une solution appartenant aux pentes ou aux abords de cette crevasse. Or on a aucun moyen de savoir « où » est cette crevasse, ni dans quelle direction se diriger pour s'en rapprocher, d'autant plus qu'on peut à tout moment se retrouver piégé dans une autre crevasse moins profonde et y rester.

Stratégie n°1 : Considérer que le voisinage d'une solution de bonne qualité comporte probablement d'autres solutions de bonne qualité. Il faudrait donc peaufiner les algorithmes de construction initiale pour que la recherche locale puisse partir sur des bonnes bases. D'expérience, ce n'est finalement pas si important que ça : Les résultats finaux après optimisation de solutions construites par un bon constructeur (plus proche voisins) et par le pire (plus lointain voisin) sont presque équivalents avec mon code. Les nombreux passages dans mes algorithmes d'optimisation finissent par lisser les différences initiales de qualité.

Stratégie n°2 : Multiplier le nombre de solutions initiales et y appliquer à chacune une recherche locale. On fait ainsi grimper la probabilité qu'une solution initiale « atterrisse » à proximité de la crevasse la plus profonde et nous fasse ainsi accéder au minimum global. Cette stratégie donne de plutôt bons résultats, mais trouver le minimum global reste tout de même quasiment impossible au vu de l'étendue de l'espace des solutions.

Stratégie n°3 : Pour tenter d'améliorer une solution en sortie de recherche locale (qui est donc coincée dans un minimum local), on peut tenter d'accepter des perturbations pas forcément améliorantes. On donne ainsi l'occasion à une solution de « s'extirper » d'une crevasse moyennement profonde afin de pouvoir atterrir à proximité d'une crevasse potentiellement plus profonde et d'ainsi y accéder lors d'une nouvelle recherche locale.

Enfin, la stratégie qui m'aura permis de produire mes meilleurs résultats : mixer savamment les 3 stratégies précédentes, ajuster les paramètres en fonction de la taille de l'instance, lancer l'exécution et espérer que le temps de calcul ne sera pas trop long (au-delà de 10 heures d'exécution y en a marre !).

E. VOTRE MISSION, SI VOUS L'ACCEPTEZ …

Notre objectif durant ce TP, va donc être de réaliser un programme permettant de :

- Lire un fichier contenant les données d'instanciation d'un VRP (Problème de Routage de Véhicules) et, par extension, d'un TSP (Problème du Voyageur de Commerce).
- Construire des solutions initiales de plus ou moins bonne qualité.
- Calculer le coût d'une solution quelconque donnée.
- En partant d'une solution quelconque, trouver son minimum local.
- Répéter ce processus avec un nombre donné (paramétrable) d'autres solutions afin d'explorer au maximum l'espace des solutions.
- Sélectionner les meilleures solutions trouvées précédemment et tenter de leur faire changer avantageusement de minimum local en acceptant des perturbations dévalorisantes.
- Régler, théoriquement et surtout empiriquement, les différents paramètres de façon à trouver les meilleures solutions possible tout en conservant un temps de calcul raisonnable.

La suite du rapport va s'atteler à essayer de décrire le plus clairement possible la façon dont je m'y suis pris.

III. Tour d'horizon de mon code source

A. CHOIX TECHNIQUE ET STRUCTURE DU PROJET

J'ai débuté le projet sous Visual Studio 2019 en codant en C, avec l'appui de certains éléments du C++ (par exemple std::string pour simplifier la lecture des fichiers) et en allouant statiquement les tableaux dans mes structures de données. Cela m'a coûté beaucoup de temps en débogages divers, variés et souvent peu intuitifs dus à des petites différences de fonctionnement entre le C et le C++ (Cela dit, j'aurais pu m'y attendre). En plus de ça je me suis rendu compte (plutôt tardivement) que l'allocation statique saturait la pile (stack) à partir d'une certaine taille d'instance et qu'il fallait donc que je déplace urgemment mes données d'exécution vers le tas (heap) avec de l'allocation dynamique sous peine de devoir continuer à résoudre des bugs incompréhensibles. J'ai donc naïvement commencé à transformer mon code en ajoutant des malloc() et des free() partout mais ça a résulté en bugs sur les variables de type std::string. Merci à l'équipe d'isilabs qui a identifié, non sans peine, la source du problème et qui m'a expliqué qu'il fallait que je remplace « malloc() » par « new » et « free() » par « delete » (malloc() et free() : C ; new et delete : C++) pour des raisons obscures d'appel automatique de constructeur pour les variables de type std::string.

Tout cela m'a conduit à prendre la résolution, après la fin de ce TP, de ne plus jamais mixer deux langages différents dans le même code quand ce n'est pas fait pour, même s'ils se ressemblent beaucoup. En parallèle, j'ai décidé d'abandonner Visual studio 2019 sur Windows et la compilation automatique pour passer à Visual Studio Code sur Linux et ajouter mon propre makefile. J'ai ainsi acquis bien plus de contrôle sur la situation et de confort, mais ce sont des préférences personnelles. (Bon, je dis ça mais

au final j'ai tout de même écrit une version sur Visual Studio 2019 pour les amateurs de Windows. En revanche, je la déconseille car la version Linux reste plus fournie et plus travaillée en comparaison.)

Finalement, je me suis retrouvé avec 2 modules en plus de mon fichier main.cpp :

- outils.cpp agrémenté de son entête outils.h
- allocDynamiqueEtAffichage.cpp avec son entête allocDynamiqueEtAffichage.h

Le tout accompagné d'un fichier d'entête general.h comportant les définitions des paramètres (constantes de préprocesseur) et des structures de données utilisées partout dans le code.

Pour compiler sous Linux, il suffit d'exécuter dans un terminal la commande « make » dans le répertoire « Code source Linux ». L'exécution se fera avec la commande « ./projetVRP » dans ce répertoire. Vous retrouverez, toujours dans ce répertoire, un README comportant plus de détails et des trucs et astuces par rapport à la compilation/exécution sous Linux.

B. PRINCIPALES STRUCTURES DE DONNEES ET ALLOCATION DYNAMIQUE

J'ai déclaré, dans general.h, 5 structures de données différentes avec la commande « typedef struct » (C'est une commande C alors que j'aurais tout autant pu déclarer des classes C++ avec « class MaClasse {} », c'est une des nombreuses incohérences de mon code dues au mix du C et du C++)

- **instance_t** pour stocker les données d'une instance : le nombre de sommets (villes), la capacité des véhicules, un tableau à deux dimensions répertoriant les distances séparant les sommets et un tableau à une dimension pour préciser la quantité demandée à chaque étape.
- **solutionTSP_t** pour stocker les données d'une solution d'un TSP : le nom de cette solution (généralement hérité du nom de l'algorithme de construction initiale), le coût de cette solution, la quantité transportée, le nombre de sommets visités (sans compter le dépôt), un tableau à une dimension énumérant l'ordre des étapes (dépôts inclus).
- **solutionVRP_t** pour stocker les données d'une solution du VRP : le nom de cette solution (généralement hérité du nom de l'algorithme de construction initiale), le coût de cette solution, la quantité transportée, le nombre de tournées (solution d'un TSP) qui composent cette solution du VRP, un tableau à une dimension de solutionTSP_t comportant les données de chacune des tournées.
- **contenantTableauSolutionsTSP_t** pour stocker un ensemble de tours géants (solutionTSP_t) : le nom de ce contenant (généralement la prochaine ou la précédente opération effectuée sur cet ensemble), le nombre de tours géants qu'il contient, un tableau à une dimension de solutionsTSP_t comportant les données de chacun des tours géants.
- **contenantTableauSolutionsVRP _t** pour stocker un ensemble de solutions du VRP (par exemple pour sauvegarder les meilleures solutions du VRP trouvées) : le nom du contenant, le nombres de solutions du VRP qu'il contient, un tableau à une dimension de solutionVRP_t comportant les données de chacune de ces solutions.

Figure 5 : Captures d'écran dans general.h → Mes différentes structures de données

```
typedef struct typeContenantTableauSolutionsVRP
{
    std::string nomTableauSolutionsVRP;
    int nbrSolutionsVRP;
    solutionVRP_t * tableauSolutionsVRP;
}contenantTableauSolutionsVRP_t;

typedef struct typeContenantTableauSolutionsTSP
{
    std::string nomTableauSolutionsTSP;
    int nbrSolutionsTSP;
    solutionTSP_t * tableauSolutionsTSP;
}contenantTableauSolutionsTSP_t;

typedef struct typeSolutionTSP
{
    std::string nomSolutionTSP;
    int coutTotalTSP;
    int quantityTotaleTSP;
    int nbrNoeudsTSP;
    int * etapesTSP;
}solutionTSP_t;

typedef struct typeSolutionVRP
{
    std::string nomSolutionVRP;
    int nbrTourneesVRP;
    int coutTotalVRP;
    int quantityTotaleVRP;
    solutionTSP_t * tableauTSP;
}solutionVRP_t;

typedef struct typeInstance
{
    int nbrNoeudsInstance;
    int capacityVehicle;
    int ** distances;
    int * quantityCollectable;
}instance_t;
```

J'ai été obligé d'allouer la plupart de mes tableaux dynamiquement (comme en témoignent les pointeurs dans la Figure 5 ci-dessus) pour des raisons de saturation. J'ai donc, pour chaque type de structure de données, fait une fonction d'allocation et une fonction de libération (voir Figure 6 ci-dessous). Ces fonctions ont toutes le même nom, mais n'ont pas le même prototype : ce sont donc des overload (surcharge). Le cycle de vie d'une instance d'une de mes structures de données est donc obligatoirement marqué par les commandes suivantes, pour éviter les dysfonctionnements et les fuites de mémoire :

- ******_t nomStructure ;
- fonctionMalloc(nomStructure) ;
- *utilisation de la structure nomStructure*
- fonctionFree(nomStructure);

```
void fonctionMalloc(instance_t& instance);
void fonctionMalloc(solutionTSP_t& solutionTSP);
void fonctionMalloc(solutionVRP_t& solutionVRP);
void fonctionMalloc(contenantTableauSolutionsTSP_t& contenantTableauSolutionsTSP, int nbrSolutionsTSPprevu);
void fonctionMalloc(contenantTableauSolutionsVRP_t& contenantTableauSolutionsVRP, int nbrSolutionsVRPprevu);

void fonctionFree(instance_t& instance);
void fonctionFree(solutionTSP_t& solutionTSP);
void fonctionFree(solutionVRP_t& solutionVRP);
void fonctionFree(contenantTableauSolutionsTSP_t& contenantTableauSolutionsTSP, int nbrSolutionsTSPprevu);
void fonctionFree(contenantTableauSolutionsVRP_t& contenantTableauSolutionsVRP, int nbrSolutionsVRPprevu);
```

Figure 6 : Capture d'écran dans allocDynamiqueEtAffichage.h → Fonctions d'allocation et de libération de mémoire pour différentes structures de données

C. MES PRINCIPALES FONCTIONS

```
// PRINCIPALES FONCTIONS

void instanciation(std::string nomFichier, instance_t& instance);


void evaluerTSP(instance_t instance, solutionTSP_t& solutionTSP, bool affichage);
void evaluerVRP(instance_t instance, solutionVRP_t& solutionVRP, bool affichage);


void productionDeToursGeantsPourSPLIT(contenantTableauSolutionsTSP_t& contenantTableauSolutionsTSPpourSPLIT, instance_t instance);
    void constructionTSPplusProcheVoisins(instance_t instance, solutionTSP_t& solutionTSP);
    void constructionTSPplusProchesVoisinsRandomized(instance_t instance, solutionTSP_t& solutionTSP);
    void constructionTSPavecTousLesSommetsIndiceCroissant(instance_t instance, solutionTSP_t& solutionTSP);
    void constructionTSPcheapestInsertion(instance_t instance, solutionTSP_t& solutionTSP);
    void constructionTSPplusLointainVoisin(instance_t instance, solutionTSP_t& solutionTSP);
    void constructionTSPrandom(instance_t instance, solutionTSP_t& solutionTSP);

    void RL_2optTSP(instance_t instance, solutionTSP_t& solutionTSP);
    void RL_deplacementSommetTSP(instance_t instance, solutionTSP_t& solutionTSP);


void productionDeVRPpour1erCycleOpti(contenantTableauSolutionsTSP_t& , contenantTableauSolutionsVRP_t& , instance_t instance);
    void SPLIT(instance_t instance, solutionTSP_t& solutionTSP, solutionVRP_t& solutionVRP);
    void constructionVRPsommetsLointainsPuisCheapestInsertion(instance_t instance, solutionVRP_t& solutionVRP);


void procedure1erCycleOptimisation(instance_t instance, contenantTableauSolutionsVRP_t , contenantTableauSolutionsVRP_t& );
    void procedureOptiCompleteSolutionVRP(instance_t instance, solutionVRP_t& solutionVRP);
        void RL_2optVRP(instance_t instance, solutionVRP_t& solutionVRP);
        void RL_deplacementSommetVRP(instance_t instance, solutionVRP_t& solutionVRP);
        bool fusionTourneesDansVRP(instance_t instance, solutionVRP_t& solutionVRP);


void procedure2emeCycleOptimisation(instance_t instance, contenantTableauSolutionsVRP_t , contenantTableauSolutionsVRP_t& );
    void effectuerRegressionRecursive(instance_t instance, solutionVRP_t , solutionVRP_t& , int profondeurActuelle, bool & continuer);
      void effectuerRegression(instance_t instance, solutionVRP_t solutionVRP);
```

Figure 7 : Capture d'écran dans outils.h → Mes principales fonctions à peu près dans l'ordre d'appel par le main() (Une fonction avec d'autres fonctions en retrait en-dessous d'elle signifie qu'elle est la principale exploitante de ces fonctions).

Globalement, l'ordre des fonctions respecte l'ordre dans lequel elles sont appelées dans le main(). Malgré les noms de fonctions assez explicites de la Figure 7 ci-dessus, je tiens à préciser quelques détails :

Les 4 recherches locales différentes proviennent de 2 stratégies différentes (RL_2opt : permutation et/ou inversion ; RL_deplacementSommet : déplacement de sommets) ainsi que 2 structures de données différentes auxquelles appliquer la recherche locale (solution d'un TSP et solution du VRP). On exécute donc uniquement la fonction SPLIT() sur des tours géants optimisés au maximum (ou non, c'est selon le paramètre GROSSEOPTI_TSP_INITIALE) → **Stratégie n°1.**

J'ai également fait un algorithme de construction de solution du VRP (constructionVRPsommetsLointainsPuisCheapestInsertion()), mais il n'est malheureusement pas aussi performant que je ne l'espérait lors de sa création ☹ (ses résultats sont généralement d'un peu moins bonne qualité par rapport aux autres).

Le 1er cycle d'optimisation (procedure1erCycleOptimisation()) correspond en quelque sorte à la **Stratégie n°2** dans le sens où son fonctionnement se résume à multiplier les recherches locales sur un petit ensemble de solutions du VRP fraîchement SPLITées. Mais cela revient tout de même à faire

beaucoup d'essais différents pour augmenter la probabilité de trouver une solution de bonne qualité (ce qui est l'essence même de la stratégie n°2) car mes fonctions de recherche locale ont une part de hasard et il est quasiment impossible de tomber sur la même solution d'une exécution à l'autre avec la même solution initiale.

Le 2ème cycle d'optimisation (procédure2emeCycleOptimisation()) sélectionne les meilleures solutions trouvées avec le 1er cycle et tente de leur faire changer de façon avantageuse de minimum local en acceptant des perturbations dévalorisantes (**Stratégie n°3**). Cette stratégie étend en quelques sortes la portée du voisinage atteignable par la perturbation d'une solution. Elle utilise pour cela une fonction récursive entièrement paramétrable (pour éviter les trop longues exécutions) qui passe de voisins en voisins, pour au final se retrouver à explorer des zones assez lointaines de la solution initiale.

IV. Description algorithmique

A. INSTANCIATION DEPUIS UN FICHIER.TXT

Le logiciel prend en entrée un fichier .txt comme celui de la figure 8 ci-dessous par exemple. On y retrouve en premier lieu le nombre de sommets de l'instance, sans compter le dépôt (ici : N = 19, il y a donc 20 sommets en comptant le dépôt). Ensuite vient la quantité maximale que peut embarquer un véhicule puis, sur la deuxième ligne, l'indice du sommet correspondant au dépôt.

Les N+1 lignes allant de la ligne 3 à la ligne N+3 constituent en fait les entrées d'un tableau à deux dimensions contenant les distances séparant chacune des villes tel que la distance entre deux villes d'indice i et j (i et j appartenant à l'intervalle [0 ; N]²) est égale à la valeur de la ligne i et de colonne j ainsi qu'à la valeur de ligne j et de colonne i.

```
19 105
0
0       1238    2398    2183    3214    2930    3709    2627    2677    4229    4627    7891    6341    5047    8006    8159    6765    5946    6358    7358
1238    0       1494    2632    3662    3379    4158    2981    2021    2146    3635    8387    6790    5496    8455    8608    4640    3305    4275    6364
2398    1494    0       1601    2796    3238    5164    5015    4078    2793    3192    7260    5882    5355    9456    9609    8215    4510    4922    5921
2183    2632    1601    0       1891    2612    4538    4389    5695    4410    2728    8321    4734    4728    8829    8982    7588    6127    6539    10465
3214    3662    2796    1891    0       2215    4969    5555    5706    5101    3516    8540    2461    3515    5884    10149   8755    6818    7230    10685
2930    3379    3238    2612    2215    0       3473    3747    4835    5363    5762    10464   4635    2941    3534    8340    6946    7080    7492    12609
3709    4158    5164    4538    4969    3473    0       2414    4284    7443    7841    11105   6156    3717    2665    5409    5614    6987    9572    10572
2627    2981    5015    4389    5555    3747    2414    0       2529    4682    7534    10798   9248    5936    5609    5163    3257    5280    6999    10265
2677    2021    4078    5695    5706    4835    4284    2529    0       2262    4024    8782    8279    6985    8671    12308   3795    2634    4353    6759
4229    2146    2793    4410    5101    5363    7443    4682    2262    0       2963    7715    7088    6674    10782   13585   6304    3788    2695    5692
4627    3635    3192    2728    3516    5762    7841    7534    4024    2963    0       4835    6483    9225    16701   14517   7359    4843    4670    2912
7891    8387    7260    8321    8540    10464   11105   10798   8782    7715    4835    0       5588    9427    13244   16886   16800   13183   8395    5501
6341    6790    5882    4734    2461    4635    6156    9248    8279    7088    6483    5588    0       3283    6031    13975   11825   16986   12198   9304
5047    5496    5355    4728    3515    2941    3717    5936    6985    6674    9225    9427    3283    0       3751    9949    14877   8610    15703   12809
8006    8455    9456    8829    5884    3534    2665    5609    8671    10782   16701   13244   6031    3751    0       4242    7261    15593   20098   17203
8159    8608    9609    8982    10149   8340    5409    5163    12308   13585   14517   16886   13975   9949    4242    0       5068    9132    13855   17429
6765    4640    8215    7588    8755    6946    5614    3257    3795    6304    7359    16800   11825   14877   7261    5068    0       5031    10025   13600
5946    3305    4510    6127    6818    7080    6987    5280    2634    3788    4843    13183   16986   8610    15593   9132    5031    0       3757    10358
6358    4275    4922    6539    7230    7492    9572    6999    4353    2695    4670    8395    12198   15703   20098   13855   10025   3757    0       5639
7358    6364    5921    10465   10685   12609   10572   10265   6759    5692    2912    5501    9304    12809   17203   17429   13600   10358   5639    0
1       25
2       33
3       5
4       75
5       105
6       23
7       87
8       9
9       14
10      24
11      14
12      25
13      10
14      35
15      23
16      43
17      38
18      31
19      71
```

Figure 8 : Capture d'écran de HVRP_DLP_75.txt, l'instance la plus petite du lot (19 sommets)

Les N lignes allant de la ligne N + 3 + 1 à la ligne 2*N + 3 listent les quantités de marchandises demandées par chaque ville. L'indice représentant le dépôt n'y figure pas car le dépôt ne se fait pas livrer comme les autres villes.

La fonction instanciation() va donc lire ce fichier dans l'ordre d'apparition des valeurs et va remplir une structure de type instance_t (prise en argument) avec ces valeurs.

B. CONSTRUCTION D'UNE SOLUTION

Certaines solutions initiales se laissent plus facilement améliorer que d'autres et je n'ai pas encore trouvé de lien de cause à effet solide et fiable. J'ai donc écrit plusieurs fonctions de construction initiale de solution du TSP et une seule fonction de construction de solution du VRP afin de multiplier les stratégies de construction, en espérant que certaines se laisseront plus facilement optimiser que d'autres :

- constructionTSPplusProchesVoisins() : Cette fonction part du dépôt, trouve la ville la plus proche et la place en tant que 1ère étape. Ensuite elle trouve la ville voisine la plus proche, la place en tant que 2ème étape, et ainsi de suite jusqu'à ce qu'elle soit passée par toutes villes. Elle place alors le dépôt en étape finale. Un tour géant d'assez bonne qualité est créé ! Cependant, il reste identique d'une exécution à l'autre et il « oublie » parfois certains sommets, qui n'ont pas été des voisins suffisamment proches pour être sélectionné au bon moment. Le véhicule devra donc « retourner en arrière » pour repasser par ces sommets avant de finir sa tournée, ce qui est bien moins rentable que s'il avait fait un petit détour sur son chemin au bon moment.
- constructionTSPplusProchesVoisinsRandomized() : Très similaire à la fonction précédente, cette fonction choisit par contre à chaque nouvelle étape entre les NBRPLUSPROCHESVOISINS plus proches voisins, dans l'ordre de proximité avec une probabilité de PROBAPREMIERCHOIX (constantes de préprocesseur définies dans general.h) pour le sommet le plus proche, une probabilité de PROBAPREMIERCHOIX² pour le deuxième sommet le plus proche et ainsi de suite… Cela favorise donc parfois les petits détours rentables et le résultat n'est pas le même d'une exécution à l'autre. Les résultats de cette fonction sont parfois meilleurs et parfois moins bons que ceux de la fonction précédente.
- constructionTSPcheapestInsertion() : Stratégie très intéressante qui consiste à tirer une première ville au hasard et à considérer l'aller-retour (2 arêtes) que cela crée entre le dépôt et cette ville. A partir de là, la fonction va calculer, pour chacune des villes et pour chacune des arêtes (nombre de sommets insérés + 1) le coût qu'induirait un détour par cette ville sur cette arête. La fonction va ensuite sélectionner l'insertion la moins coûteuse avant de l'appliquer. Puis elle refait tous les calculs, sélectionne à nouveau et ainsi de suite jusqu'à ce que tous les sommets aient été sélectionnés. Cette construction donne de plutôt bons résultats.
- constructionTSPplusLointainVoisin() : Cette fonction a la particularité de construire la PIRE solution possible avec un procédé très simple : C'est une belle ironie quand on sait que la meilleure solution est quasiment impossible à trouver à partir d'une certaine taille d'instance. Cette fonction reprend le même fonctionnement que constructionTSPplusProchesVoisins() à la différence près qu'elle sélectionne à chaque étape le sommet le plus éloigné au lieu du plus proche. Cette différence d'exactement un caractère (un > au lieu d'un < dans une sous-fonction de tri) change du tout au tout la nature du résultat, étonnant non ? Malgré tout, les résultats de cette fonction restent acceptables après optimisation. Personnellement, cela me laisse bouche bée.

- constructionTSPavecTousLesSommetsIndiceCroissant() : Fonction extrêmement bête qui classe les étapes par ordre d'indice des villes. Elle va donc d'abord passer par la ville d'indice 1, puis par celle d'indice 2 et ainsi de suite, sachant que les indices n'ont rien à voir avec la distance entre les villes.
- constructionTSPrandom() : Idem, à chaque étape, la prochaine sera tirée au sort entre toutes les étapes restantes, c'est tout aussi peu productif. Bien que cette construction produit, tout comme la précédente, des résultats médiocres, elles permettent tout de même de multiplier les philosophies de construction. On peut ainsi peut-être espérer créer d'autres opportunités pour arriver, à terme, au minimum global. Cela m'a également permis de constater que la qualité des fonctions de constructions initiales est bien moins importante que la qualité des fonctions d'optimisation car les résultats provenant de stratégies de constructions initiales absurdes étaient quand même acceptables après optimisation.
- constructionVRPsommetsLointainsPuisCheapestInsertion() : C'est la seule fonction de construction de solution du VRP, les autres donnant toutes des solutions du TSP qui se font ensuite SPLITer. La réalisation de cette fonction m'a pris pas mal de temps pour finalement être décevante par rapport à mes attentes, bien qu'elle ait un fonctionnement plutôt complexe. C'est pour cela que je ne vais pas perdre trop de temps en descriptions, tout autant inutiles que l'acharnement dont j'ai fait preuve lors de sa réalisation. Grossièrement, cette fonction essaie dans un premier temps « d'éliminer » les sommets les plus excentrés par rapport au dépôt en créant plusieurs tournées différentes. Pour chacune de ces tournées, la fonction sélectionne le sommet pas encore visité le plus éloigné du dépôt et fait un aller-retour entre le dépôt et ce sommet. Ensuite, elle applique la stratégie de l'insertion la moins coûteuse jusqu'à ce qu'elle soit limitée par la quantité maximale transportable. Ainsi de suite jusqu'à un certain moment, où la fonction va considérer qu'elle a d'ores et déjà éliminé les villes les plus excentrées et qu'elle va décider de commencer à se concentrer sur les sommets qui demandent le plus de quantité. De même, elle va appliquer la stratégie de l'insertion la moins coûteuse à partir de ces sommets. Il y a en plus des notions d'ellipses et de notes pour chaque sommet avec des coefficients de pondération mais c'est trop inutile d'expliquer car les résultats de cette fonction ne font presque jamais partie des meilleurs. Encore moins après optimisation car les solutions issues de cette construction ne se laissent apparemment pas facilement améliorer…

C. EVALUATION D'UNE SOLUTION

J'utilise pour cela, en fonction du type de la solution à évaluer, les fonctions evaluerTSP() et evaluerVRP(). Elles additionnent simplement les distances ainsi que les quantités demandées entre chaque étape. Tout au long du programme, il est important d'exécuter ces fonctions dès la modification d'une solution. Je l'utilise beaucoup au sein des fonctions de recherche locale pour tester la qualité d'une solution fille et déterminer si elle est améliorante par rapport à la solution mère ou non.

D. RECHERCHE D'UN MINIMUM LOCAL

Encore une fois, pour la résolution de notre fameux VRP, rechercher ou du moins se rapprocher du minimum global doit en premier lieu passer par la recherche intensive d'un grand nombre de minimas locaux. Concrètement, dans mon code, c'est plutôt la fonction procedureOptiCompleteSolutionVRP () qui s'en occupe. Elle appelle plusieurs fois les sous-fonctions RL_2opt(), RL_deplacementSommetVRP() et fusionTourneesDansVRP() qui s'occupent de trouver un minimum local dont l'étendue du voisinage est limitée à une et une seule perturbation au choix parmi les suivante : permutation de 2 sommets, inversion, déplacement d'un sommet. Il est possible de créer une recherche locale avec une notion plus

étendue de voisinage, comme par exemple un voisinage atteignable par une combinaison de plusieurs perturbations ou alors par des perturbations avec des mécaniques plus élaborées. Les résultats obtenus y gagneraient en qualité mais les temps de calculs deviendraient immédiatement plus longs car la recherche locale arpenterait une « plus vaste localité ». Cependant, le deuxième cycle d'optimisation suit plus ou moins ces idées.

La Figure 9 illustre encore une fois l'intérêt d'un algorithme de descente lors d'une recherche locale. On voit également que la stratégie consistant en la multiplication des solutions initiales a été d'une efficacité redoutable et a permis de trouver le minimum global de l'Espace des Solutions.

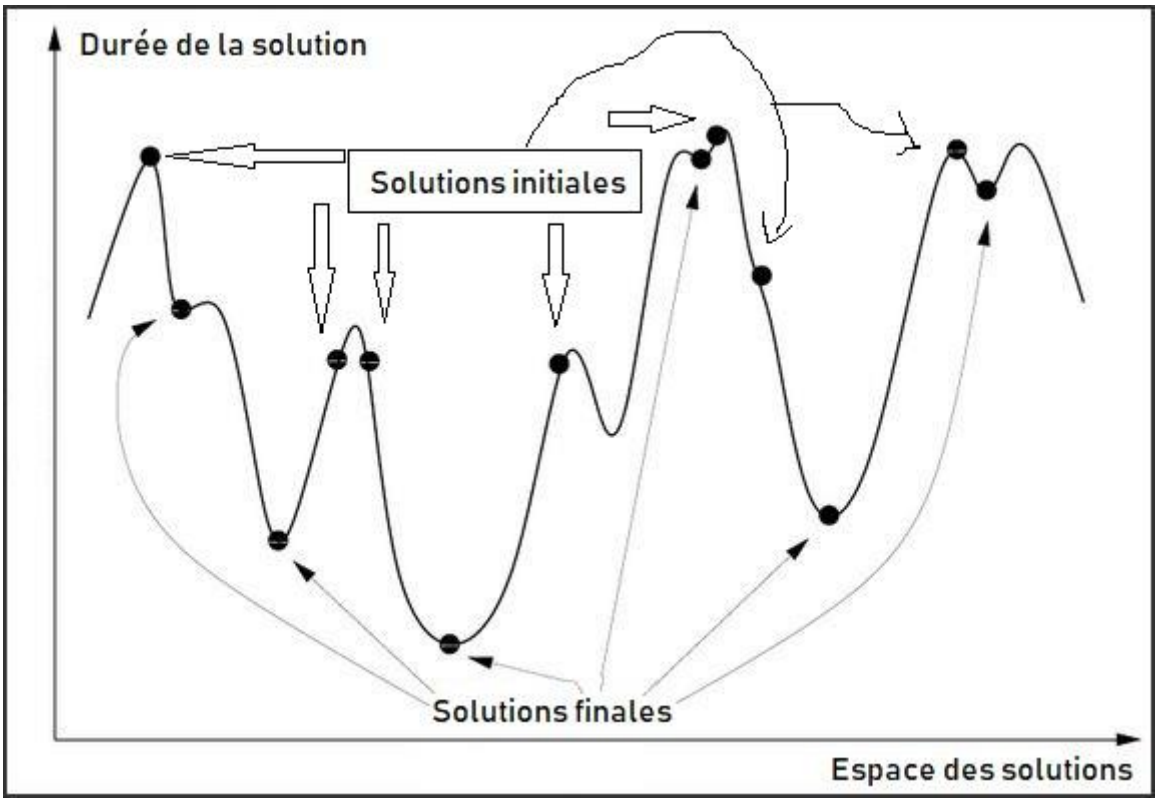


Figure 9 : Illustration des différentes entrées et sorties possibles d'une Recherche Locale

E. RECHERCHE DU MINIMUM GLOBAL

J'ai pendant longtemps voulu adapter le principe de recuit simulé pour diriger ma recherche de minimum global, comme détaillé sur cette page Wikipédia. Je me suis ensuite rendu compte que ce n'était pas vraiment adapté à mon problème ainsi qu'à ma façon de faire. Je me suis donc tourné vers une sorte « d'explorateur récursif de voisinage » que j'ai surnommé « 2ème cycle d'optimisation ».

Ce cycle effectue différentes explorations de l'espace des solutions grâce à la fonction récursive effectuerRegressionRecursive(). Une exploration est déclenchée par un appel initial de cette fonction. Lors de cet appel, on passe en paramètre la solution que l'on souhaite optimiser : c'est ce qu'on va appeler la solution initiale. L'appel initial de cette fonction applique une première perturbation aléatoire à la solution initiale, puis lance la fonction procedureOptiCompleteSolutionVRP() qui est une recherche locale tentant de trouver le minimum de la nouvelle localité de la solution initiale fraîchement perturbée. Cette solution nouvellement créée est « la solution de profondeur 0 ». Une fois cela fait, la fonction va s'appeler elle-même NBRBRANCHESREGRESSIONRECURSIVE fois en passant

en paramètre la solution de profondeur 0. Tous ces appels agissent de façon indépendante, ils s'occupent tous de créer leur propre solution de profondeur 1 en appliquant une nouvelle perturbation suivie par une optimisation. Ensuite, si la profondeur de leur solution est strictement inférieure au paramètre PROFONDEURREGRESSIONRECURSIVE, alors elles s'appellent elles-mêmes à leur tour NBRBRANCHESREGRESSIONRECURSIVE fois en passant leur propre solution de profondeur 1 en paramètre et ainsi de suite… La fin de l'exploration est marquée par la découverte d'une solution améliorante par rapport à la solution initiale ou alors, à défaut, lorsque (A^0 + A^1 + A^2 + … + A^B) (avec A = NBRBRANCHESREGRESSIONRECURSIVE et B = PROFONDEURREGRESSIONRECURSIVE) appels ont été réalisés.

Sur la Figure 10, une solution améliorante a été trouvé lors du 13ème appel (le dernier). La fonction s'est

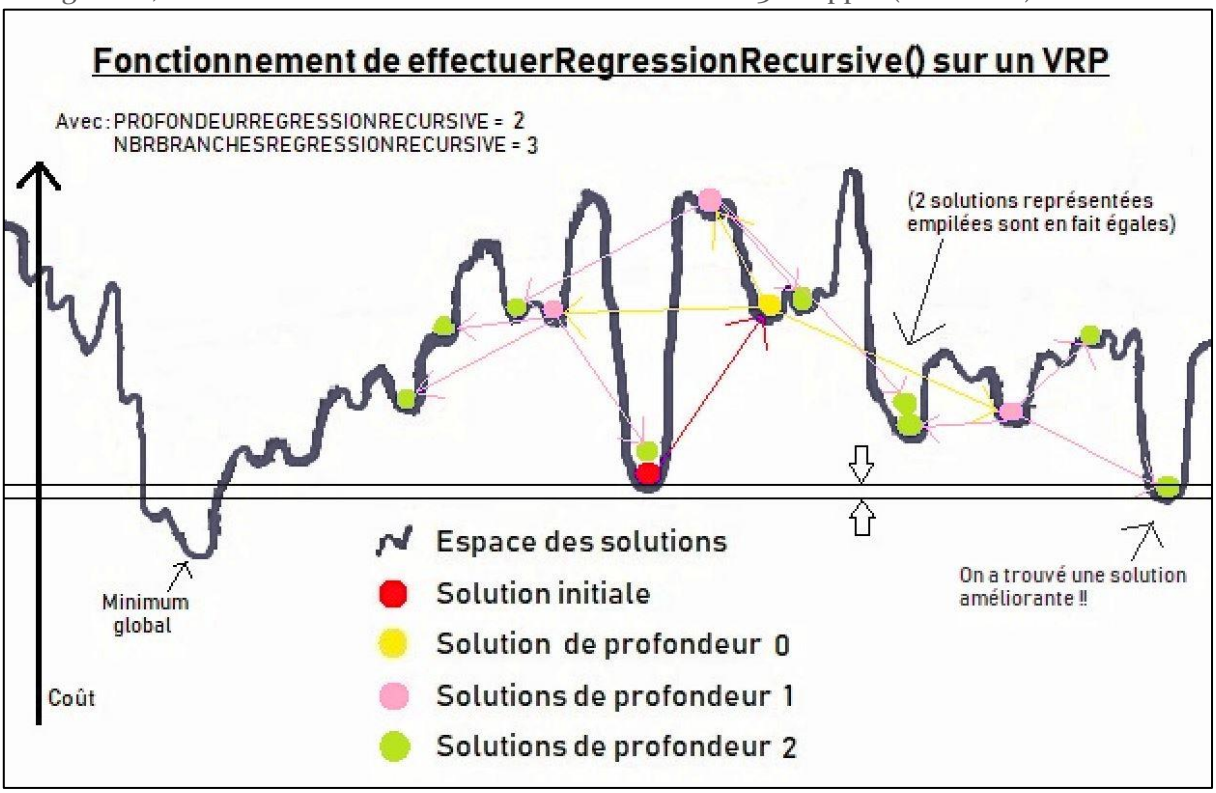


Figure 10 : Schéma explicatif du fonctionnement de effectuerRegressionRecursive() sur une solution d'un VRP.

donc arrêtée pour 2 raisons : 1+3+9 appels ont été réalisés et une solution améliorante vient d'être trouvée. On peut aussi supposer que le minimum global n'a pas été trouvé pour plusieurs causes : peut-être que les régressions successives sont majoritairement parties dans la « mauvaise direction ». Effectivement, si par exemple la solution de profondeur 0 était « allée vers la gauche » lors de l'appel initial, alors les solutions de profondeur 1 auraient été en moyenne bien moins éloignées du minimum global et une solution de profondeur 2 aurait probablement finie par le trouver. Aussi, augmenter le paramètre PROFONDEURREGRESSIONRECURSIVE aurait probablement fait en sorte que des solutions de profondeur 4 (ou même 3 avec un peu de chance) découlant de la solution de profondeur 2 la plus à gauche finissent par tomber dans la crevasse contenant le minimum global.

Pour permettre l'amélioration successive d'une solution, le 2ème cycle d'optimisation fait NBRDERUNSREGRESSIONRECURSIVE appels à la fonction effectuerRegressionRecursive(). Lors du premier appel, on passe la solution que l'on souhaite optimiser (nous appellerons cette version de la solution la solution de base) en paramètre. Lors de l'appel suivant, on passe la solution de base

agrémentée de sa modification améliorante trouvée précédemment et ainsi de suite. La solution de base se retrouve donc de plus en plus optimisée et c'est au terme de ce 2ème cycle que les meilleurs résultats ont été produits lors de mes diverses expériences.

L'exemple de la figure 10 nous montre encore une fois les effets, parfois inespérés, parfois dévastateurs du hasard : Admettons que cette situation ait été engendrée par l'exécution consécutive à l'appel initial n°8 de effectuerRegressionRecursive() (alias exécution n°8). La solution initiale de la figure est donc en fait la solution de base agrémentée d'environ 7 modifications améliorantes. Nous nous apprêtons maintenant à effectuer l'appel initial n°9 de effectuerRegressionRecursive() avec la solution de profondeur 2 la plus à droite comme solution initiale. Eh bien théoriquement, même si en réalité nous avons aucuns moyens de l'obtenir, la représentation graphique de l'espace des solutions nous informe que cette solution initiale est plus éloignée du minimum global que la solution initiale passée en paramètre de l'appel initial n°8. Trouver le minimum global lors de l'exécution n°9 sera donc moins probable que lors de l'exécution n°8.

Cela pose la question (je ne peux y répondre) de la rentabilité de la solution améliorante trouvée lors de l'exécution n°8…

V. Quelques résultats sur les différentes instances

Voici la meilleure solution trouvée sur l'instance VRP_DLP_75 :
Programme exécuté sur l'instance 'instances/en-dessous de 100 villes/VRP_DLP_75.txt' (19 sommets et quantité max = 105)
Ici une simple optimisation par permutations suffit à trouver le minimum global, comme le montre les résultats suivants :

Voici contenantTableauSauvegardeMeilleuresSolutionsVRPpour2emeCycle :
Coût total solutionVRPplusProcheVoisin : 100437; nbr tournées : 7; quantités : 690
Coût total solutionVRPplusProchesVoisinsRandomized_inverse : 100437; nbr tournées : 7; quantités : 690
Coût total solutionVRPavecTousLesSommetsIndiceCroissant_inverse : 100437; nbr tournées : 7; quantités : 690
Coût total solutionVRPrandom_inverse : 100437; nbr tournées : 7; quantités : 690
Coût total solutionVRPplusProcheVoisin : 100437; nbr tournées : 7; quantités : 690
Coût total solutionVRPplusProchesVoisinsRandomized_inverse : 100437; nbr tournées : 7; quantités : 690

Voici le détail de la meilleure solution VRP trouvée avec ses solutions TSP (tournées) successives :
Coût total = 100437 ; nbr tournées : 7; quantités : 690

Tournée d'indice 0 --> Nombre de sommets = 1; Coût Total = 5860 et Quantité Collectée = 105 : 0 5 0
Tournée d'indice 1 --> Nombre de sommets = 5; Coût Total = 25955 et Quantité Collectée = 105 : 0 2 11 12 13 6 0
Tournée d'indice 2 --> Nombre de sommets = 3; Coût Total = 24081 et Quantité Collectée = 101 : 0 16 15 14 0
Tournée d'indice 3 --> Nombre de sommets = 3; Coût Total = 15181 et Quantité Collectée = 100 : 0 3 10 19 0
Tournée d'indice 4 --> Nombre de sommets = 2; Coût Total = 8114 et Quantité Collectée = 100 : 0 1 4 0
Tournée d'indice 5 --> Nombre de sommets = 4; Coût Total = 15992 et Quantité Collectée = 92 : 0 8 17 18 9 0
Tournée d'indice 6 --> Nombre de sommets = 1; Coût Total = 5254 et Quantité Collectée = 87 : 0 7 0

Compilation effectuée le Jan 18 2023 à 17:38:09
Exécution lancée le Wed Jan 18 17:38:14 2023
Exécution achevée le Wed Jan 18 17:39:29 2023
Le programme a donc été exécuté en 75 secondes : 0 jours, 0 heures, 1 minutes et 15 secondes

Voici La meilleure solution trouvée sur l'instance VRP_DLP_01 :
Programme exécuté sur l'instance 'instances/en-dessous de 100 villes/VRP_DLP_01.txt' (91 sommets et quantité max = 275) avec les paramètres suivants :

GROSSEOPTI_TSP_INITIALE = false
NBRPREMIERCYCLE = 5
BRDEUXIEMECYCLE = 3
NBRSOLUTIONSVRPPOURDEUXIEMECYCLE = 6
NBRDERUNSREGRESSIONRECURSIVE = 15
PROFONDEURREGRESSIONRECURSIVE = 3
NBRBRANCHESREGRESSIONRECURSIVE = 7

Voici le détail de la meilleure solution VRP trouvée avec ses solutions TSP (tournées) successives :
Coût total = 1894229 ; nbr tournées = 15 ; Quantité Collectée = 4000

Tournée d'indice 0 --> Nombre de sommets = 6; Coût Total = 126136 et Quantité Collectée = 275 : 0 47 70 29 34 25 88 0
Tournée d'indice 1 --> Nombre de sommets = 7; Coût Total = 134676 et Quantité Collectée = 275 : 0 68 67 63 31 71 85 19 0
Tournée d'indice 2 --> Nombre de sommets = 11; Coût Total = 219758 et Quantité Collectée = 274 : 0 73 83 30 23 81 14 82 55 27 64 36 0
Tournée d'indice 3 --> Nombre de sommets = 9; Coût Total = 175205 et Quantité Collectée = 274 : 0 20 41 28 78 62 40 5 45 77 0
Tournée d'indice 4 --> Nombre de sommets = 3; Coût Total = 138539 et Quantité Collectée = 274 : 0 26 91 52 0
Tournée d'indice 5 --> Nombre de sommets = 5; Coût Total = 127916 et Quantité Collectée = 273 : 0 16 79 50 72 38 0
Tournée d'indice 6 --> Nombre de sommets = 9; Coût Total = 105515 et Quantité Collectée = 273 : 0 39 76 6 54 10 75 11 7 69 0
Tournée d'indice 7 --> Nombre de sommets = 4; Coût Total = 66771 et Quantité Collectée = 272 : 0 15 35 43 89 0
Tournée d'indice 8 --> Nombre de sommets = 6; Coût Total = 122345 et Quantité Collectée = 272 : 0 44 46 42 65 87 1 0
Tournée d'indice 9 --> Nombre de sommets = 4; Coût Total = 80010 et Quantité Collectée = 268 : 0 13 59 48 61 0
Tournée d'indice 10 --> Nombre de sommets = 4; Coût Total = 96060 et Quantité Collectée = 268 : 0 12 51 4 90 0
Tournée d'indice 11 --> Nombre de sommets = 4; Coût Total = 92515 et Quantité Collectée = 264 : 0 66 86 49 22 0
Tournée d'indice 12 --> Nombre de sommets = 8; Coût Total = 202076 et Quantité Collectée = 264 : 0 18 57 8 3 24 56 33 60 0
Tournée d'indice 13 --> Nombre de sommets = 8; Coût Total = 146377 et Quantité Collectée = 242 : 0 17 37 80 9 21 32 84 2 0
Tournée d'indice 14 --> Nombre de sommets = 3; Coût Total = 60330 et Quantité Collectée = 232 : 0 53 74 58 0

Compilation effectuée le Jan 18 2023 à 10:53:13
Exécution lancée le Wed Jan 18 10:53:25 2023
Exécution achevée le Wed Jan 18 12:17:51 2023
Le programme a donc été exécuté en 5066 secondes : 0 jours, 1 heures, 24 minutes et 26 secondes

Voici le meilleur résultat (je n'ai pas fait beaucoup de tests) trouvé sur l'instance VRP_DLP_38 :
Programme exécuté sur l'instance 'instances/plus de 200 villes/VRP_DLP_38.txt' (204 sommets et Quantité max = 100) avec les paramètres suivants :

GROSSEOPTI_TSP_INITIALE = false
NBRPREMIERCYCLE = 3
NBRDEUXIEMECYCLE = 2
NBRSOLUTIONSVRPPOURDEUXIEMECYCLE = 5
NBRDERUNSREGRESSIONRECURSIVE = 5
PROFONDEURREGRESSIONRECURSIVE = 3
NBRBRANCHESREGRESSIONRECURSIVE = 4

Voici le détail de la meilleure solution VRP trouvée avec ses solutions TSP (tournées) successives :
Coût total = 7812883 ; nbr tournées = 37 ; quantités = 3600

Tournée d'indice 0 --> Nombre de sommets = 5; Coût Total = 227776 et Quantité Collectée = 100 : 0 156 192 85 99 74 0
Tournée d'indice 1 --> Nombre de sommets = 4; Coût Total = 271711 et Quantité Collectée = 100 : 0 180 28 115 12 0
Tournée d'indice 2 --> Nombre de sommets = 4; Coût Total = 287262 et Quantité Collectée = 100 : 0 149 65 197 24 0
Tournée d'indice 3 --> Nombre de sommets = 8; Coût Total = 304865 et Quantité Collectée = 100 : 0 67 95 104 63 81 201 152 120 0
Tournée d'indice 4 --> Nombre de sommets = 8; Coût Total = 199067 et Quantité Collectée = 100 : 0 117 47 87 124 91 52 107 138 0
Tournée d'indice 5 --> Nombre de sommets = 11; Coût Total = 240738 et Quantité Collectée = 100 : 0 182 199 162 27 126 200 164 13 137 64 146 0
Tournée d'indice 6 --> Nombre de sommets = 4; Coût Total = 185968 et Quantité Collectée = 100 : 0 70 202 22 16 0
Tournée d'indice 7 --> Nombre de sommets = 4; Coût Total = 255227 et Quantité Collectée = 100 : 0 15 129 139 98 0
Tournée d'indice 8 --> Nombre de sommets = 4; Coût Total = 179311 et Quantité Collectée = 100 : 0 41 30 84 35 0
Tournée d'indice 9 --> Nombre de sommets = 5; Coût Total = 72581 et Quantité Collectée = 100 : 0 83 89 76 55 58 0
Tournée d'indice 10 --> Nombre de sommets = 4; Coût Total = 73373 et Quantité Collectée = 100 : 0 9 144 155 96 0
Tournée d'indice 11 --> Nombre de sommets = 8; Coût Total = 322565 et Quantité Collectée = 100 : 0 171 116 191 34 154 86 128 8 0
Tournée d'indice 12 --> Nombre de sommets = 9; Coût Total = 308768 et Quantité Collectée = 100 : 0 147 97 198 176 25 32 173 112 123 0
Tournée d'indice 13 --> Nombre de sommets = 4; Coût Total = 117337 et Quantité Collectée = 100 : 0 37 153 46 57 0
Tournée d'indice 14 --> Nombre de sommets = 5; Coût Total = 253696 et Quantité Collectée = 99 : 0 92 160 39 172 167 0
Tournée d'indice 15 --> Nombre de sommets = 6; Coût Total = 260791 et Quantité Collectée = 99 : 0 79 158 59 170 56 68 0
Tournée d'indice 16 --> Nombre de sommets = 8; Coût Total = 233611 et Quantité Collectée = 99 : 0 43 71 77 127 109 166 10 2 0
Tournée d'indice 17 --> Nombre de sommets = 10; Coût Total = 284032 et Quantité Collectée = 99 : 0 190 131 163 44 3 21 102 50 145 75 0
Tournée d'indice 18 --> Nombre de sommets = 6; Coût Total = 297665 et Quantité Collectée = 99 : 0 72 4 133 94 31 177 0
Tournée d'indice 19 --> Nombre de sommets = 5; Coût Total = 255694 et Quantité Collectée = 99 : 0 36 181 196 17 187 0
Tournée d'indice 20 --> Nombre de sommets = 4; Coût Total = 141540 et Quantité Collectée = 99 : 0 135 54 148 20 0
Tournée d'indice 21 --> Nombre de sommets = 6; Coût Total = 230204 et Quantité Collectée = 98 : 0 53 194 161 143 150 165 0
Tournée d'indice 22 --> Nombre de sommets = 4; Coût Total = 284633 et Quantité Collectée = 98 : 0 119 125 168 19 0
Tournée d'indice 23 --> Nombre de sommets = 3; Coût Total = 142189 et Quantité Collectée = 98 : 0 159 188 78 0
Tournée d'indice 24 --> Nombre de sommets = 9; Coût Total = 147566 et Quantité Collectée = 97 : 0 157 179 193 114 61 186 185 14 100 0
Tournée d'indice 25 --> Nombre de sommets = 5; Coût Total = 236802 et Quantité Collectée = 97 : 0 1 108 40 105 189 0
Tournée d'indice 26 --> Nombre de sommets = 4; Coût Total = 133483 et Quantité Collectée = 96 : 0 141 60 33 151 0
Tournée d'indice 27 --> Nombre de sommets = 1; Coût Total = 239764 et Quantité Collectée = 96 : 0 118 0
Tournée d'indice 28 --> Nombre de sommets = 6; Coût Total = 163013 et Quantité Collectée = 96 : 0 6 101 136 38 203 111 0
Tournée d'indice 29 --> Nombre de sommets = 7; Coût Total = 185608 et Quantité Collectée = 95 : 0 18 110 113 178 73 140 184 0
Tournée d'indice 30 --> Nombre de sommets = 2; Coût Total = 106601 et Quantité Collectée = 95 : 0 49 88 0

Tournée d'indice 31 --> Nombre de sommets = 6; Coût Total = 289005 et Quantité Collectée = 94 : 0 142 121 93 66 48 26 0
Tournée d'indice 32 --> Nombre de sommets = 5; Coût Total = 84482 et Quantité Collectée = 93 : 0 106 11 45 90 82 0
Tournée d'indice 33 --> Nombre de sommets = 4; Coût Total = 266570 et Quantité Collectée = 91 : 0 195 42 183 62 0
Tournée d'indice 34 --> Nombre de sommets = 5; Coût Total = 165633 et Quantité Collectée = 91 : 0 103 204 7 122 23 0
Tournée d'indice 35 --> Nombre de sommets = 5; Coût Total = 170589 et Quantité Collectée = 91 : 0 175 174 80 5 169 0
Tournée d'indice 36 --> Nombre de sommets = 6; Coût Total = 193163 et Quantité Collectée = 81 : 0 130 134 132 51 69 29 0

Compilation effectuée le Jan 18 2023 à 18:14:17
Exécution lancée le Wed Jan 18 18:14:23 2023
Exécution achevée le Wed Jan 18 18:34:38 2023
Le programme a donc été exécuté en 1215 secondes : 0 jours, 0 heures, 20 minutes et 15 secondes

Voici le résultat de la seule exécution faite sur la pire de toutes les instance (nightmare instance) : VRP_DLP_18
Programme exécuté sur l'instance 'instances/plus de 200 villes/VRP_DLP_18.txt' (255 sommets et Quantité max = 100) avec les paramètres suivants :

GROSSEOPTI_TSP_INITIALE = false
NBRPREMIERCYCLE = 3
NBRDEUXIEMECYCLE = 2
NBRSOLUTIONSVRPPOURDEUXIEMECYCLE = 5
NBRDERUNSREGRESSIONRECURSIVE = 5
PROFONDEURREGRESSIONRECURSIVE = 3
NBRBRANCHESREGRESSIONRECURSIVE = 4

Voici le détail de la meilleure solution VRP trouvée avec ses solutions TSP (tournées) successives :
Coût total = 5927813 ; nbr tournées = 51 ; quantités = 4900

Tournée d'indice 0 --> Nombre de sommets = 1; Coût Total = 74354 et Quantité Collectée = 100 : 0 26 0
Tournée d'indice 1 --> Nombre de sommets = 1; Coût Total = 155766 et Quantité Collectée = 100 : 0 233 0
Tournée d'indice 2 --> Nombre de sommets = 1; Coût Total = 125816 et Quantité Collectée = 100 : 0 123 0
Tournée d'indice 3 --> Nombre de sommets = 1; Coût Total = 77812 et Quantité Collectée = 100 : 0 178 0
Tournée d'indice 4 --> Nombre de sommets = 8; Coût Total = 198319 et Quantité Collectée = 100 : 0 243 25 14 215 138 147 99 190 0
Tournée d'indice 5 --> Nombre de sommets = 3; Coût Total = 125227 et Quantité Collectée = 100 : 0 163 21 195 0
Tournée d'indice 6 --> Nombre de sommets = 8; Coût Total = 128674 et Quantité Collectée = 100 : 0 239 116 213 198 9 94 40 58 0
Tournée d'indice 7 --> Nombre de sommets = 11; Coût Total = 142302 et Quantité Collectée = 100 : 0 194 134 231 185 64 77 201 80 150 47 118 0
Tournée d'indice 8 --> Nombre de sommets = 8; Coût Total = 206458 et Quantité Collectée = 100 : 0 237 184 68 104 65 51 175 10 0
Tournée d'indice 9 --> Nombre de sommets = 8; Coût Total = 238519 et Quantité Collectée = 100 : 0 192 44 207 205 166 222 171 45 0
Tournée d'indice 10 --> Nombre de sommets = 8; Coût Total = 170290 et Quantité Collectée = 100 : 0 114 34 120 153 151 154 70 29 0
Tournée d'indice 11 --> Nombre de sommets = 7; Coût Total = 126736 et Quantité Collectée = 100 : 0 226 55 100 84 72 252 214 0
Tournée d'indice 12 --> Nombre de sommets = 4; Coût Total = 99359 et Quantité Collectée = 100 : 0 37 132 189 2 0
Tournée d'indice 13 --> Nombre de sommets = 7; Coût Total = 109956 et Quantité Collectée = 100 : 0 247 74 32 124 105 165 169 0
Tournée d'indice 14 --> Nombre de sommets = 9; Coût Total = 141170 et Quantité Collectée = 100 : 0 161 162 42 176 121 177 240 54 117 0
Tournée d'indice 15 --> Nombre de sommets = 2; Coût Total = 149838 et Quantité Collectée = 99 : 0 79 144 0
Tournée d'indice 16 --> Nombre de sommets = 8; Coût Total = 113125 et Quantité Collectée = 99 : 0 225 218 216 228 110 28 17 208 0
Tournée d'indice 17 --> Nombre de sommets = 5; Coût Total = 71898 et Quantité Collectée = 99 : 0 111 210 149 141 188 0
Tournée d'indice 18 --> Nombre de sommets = 3; Coût Total = 139827 et Quantité Collectée = 99 : 0 235 126 96 0
Tournée d'indice 19 --> Nombre de sommets = 7; Coût Total = 111622 et Quantité Collectée = 99 : 0 98 67 12 8 33 209 131 0
Tournée d'indice 20 --> Nombre de sommets = 4; Coût Total = 78378 et Quantité Collectée = 99 : 0 63 19 52 152 0
Tournée d'indice 21 --> Nombre de sommets = 7; Coût Total = 154347 et Quantité Collectée = 99 : 0 156 60 53 157 41 18 229 0
Tournée d'indice 22 --> Nombre de sommets = 5; Coût Total = 44251 et Quantité Collectée = 99 : 0 168 186 200 223 129 0
Tournée d'indice 23 --> Nombre de sommets = 6; Coût Total = 154883 et Quantité Collectée = 98 : 0 31 73 78 242 82 87 0
Tournée d'indice 24 --> Nombre de sommets = 6; Coût Total = 182108 et Quantité Collectée = 98 : 0 49 97 172 119 86 183 0
Tournée d'indice 25 --> Nombre de sommets = 9; Coût Total = 117979 et Quantité Collectée = 98 : 0 246 219 71 15 23 20 248 173 158 0
Tournée d'indice 26 --> Nombre de sommets = 2; Coût Total = 97329 et Quantité Collectée = 98 : 0 220 253 0
Tournée d'indice 27 --> Nombre de sommets = 1; Coût Total = 75954 et Quantité Collectée = 98 : 0 245 0
Tournée d'indice 28 --> Nombre de sommets = 3; Coût Total = 75543 et Quantité Collectée = 98 : 0 206 241 128 0
Tournée d'indice 29 --> Nombre de sommets = 10; Coût Total = 219779 et Quantité Collectée = 98 : 0 136 199 62 244 217 95 1 57 204 39 0
Tournée d'indice 30 --> Nombre de sommets = 2; Coût Total = 97395 et Quantité Collectée = 98 : 0 11 22 0
Tournée d'indice 31 --> Nombre de sommets = 8; Coût Total = 206506 et Quantité Collectée = 98 : 0 27 38 30 6 115 7 106 203 0
Tournée d'indice 32 --> Nombre de sommets = 6; Coût Total = 127151 et Quantité Collectée = 98 : 0 66 36 155 170 93 224 0
Tournée d'indice 33 --> Nombre de sommets = 4; Coût Total = 79142 et Quantité Collectée = 97 : 0 59 35 230 130 0
Tournée d'indice 34 --> Nombre de sommets = 7; Coût Total = 109865 et Quantité Collectée = 97 : 0 75 109 179 5 234 160 193 0
Tournée d'indice 35 --> Nombre de sommets = 2; Coût Total = 34273 et Quantité Collectée = 97 : 0 83 139 0
Tournée d'indice 36 --> Nombre de sommets = 1; Coût Total = 86512 et Quantité Collectée = 96 : 0 50 0
Tournée d'indice 37 --> Nombre de sommets = 1; Coût Total = 137382 et Quantité Collectée = 96 : 0 174 0
Tournée d'indice 38 --> Nombre de sommets = 6; Coût Total = 125719 et Quantité Collectée = 96 : 0 212 143 43 137 76 251 0
Tournée d'indice 39 --> Nombre de sommets = 1; Coût Total = 21774 et Quantité Collectée = 95 : 0 133 0
Tournée d'indice 40 --> Nombre de sommets = 1; Coût Total = 63532 et Quantité Collectée = 95 : 0 90 0
Tournée d'indice 41 --> Nombre de sommets = 6; Coût Total = 79279 et Quantité Collectée = 95 : 0 221 108 91 227 236 249 0
Tournée d'indice 42 --> Nombre de sommets = 9; Coût Total = 157945 et Quantité Collectée = 94 : 0 13 103 107 232 61 56 92 127 81 0

Tournée d'indice 43 --> Nombre de sommets = 7; Coût Total = 132601 et Quantité Collectée = 93 : 0 112 46 187 180 254 4 159 0
Tournée d'indice 44 --> Nombre de sommets = 8; Coût Total = 92594 et Quantité Collectée = 92 : 0 238 181 101 140 250 211 122 202 0
Tournée d'indice 45 --> Nombre de sommets = 6; Coût Total = 166450 et Quantité Collectée = 92 : 0 196 88 113 191 48 167 0
Tournée d'indice 46 --> Nombre de sommets = 3; Coût Total = 66565 et Quantité Collectée = 91 : 0 69 16 135 0
Tournée d'indice 47 --> Nombre de sommets = 5; Coût Total = 62770 et Quantité Collectée = 85 : 0 197 182 24 125 3 0
Tournée d'indice 48 --> Nombre de sommets = 2; Coût Total = 40569 et Quantité Collectée = 81 : 0 102 89 0
Tournée d'indice 49 --> Nombre de sommets = 4; Coût Total = 81466 et Quantité Collectée = 74 : 0 148 255 142 164 0
Tournée d'indice 50 --> Nombre de sommets = 3; Coût Total = 50704 et Quantité Collectée = 62 : 0 145 146 85 0

Compilation effectuée le Jan 18 2023 à 18:37:43
Exécution lancée le Wed Jan 18 18:37:46 2023
Exécution achevée le Wed Jan 18 19:47:04 2023
Le programme a donc été exécuté en 4158 secondes : 0 jours, 1 heures, 9 minutes et 18 secondes

VI. Conclusion

Tout d'abord, j'aimerais identifier plusieurs points qui permettraient d'améliorer mon programme, à commencer par mon choix de générateur de nombres aléatoires. J'ai utilisé la fonction rand() de la librairie standard alors que c'est quasiment considéré comme un sacrilège depuis qu'on découvert le Mersenne Twister, en TP de simulation au premier semestre. *Message d'excuse à M. David Hill : Je vous prie de m'excuser, je ne sais pas ce qu'il m'a pris de faire ça ...*.

Contrairement à l'énoncé, j'ai toujours considéré que la flotte de véhicule est illimitée. J'ai préféré me concentrer sur des fonctionnalités plus importantes, d'autant plus que mon code cherche dans tous les cas à construire des solutions nécessitant le moins de véhicules possible. Il a par exemple trouvé sur l'instance VRP_DLP_01 des solutions nécessitant seulement 15 camions, ce qui est le nombre minimum possible pour cette instance. En parallèle, de nombreuses optimisations de code permettant de grappiller des économies de complexité de ci de là ainsi qu'une étude plus précise permettant de trouver le réglage idéal des paramètres d'exécution sont envisageables.

Une fonctionnalité supplémentaire que je trouverais intéressante : Implémenter, peut-être dans un autre langage, une représentation graphique de la meilleure solution trouvée pour une exécution, ou alors un affichage dynamique tout au long de l'exécution du programme pour voir un peu l'évolution de la solution tout au long de la construction ou de l'optimisation. On pourrait peut-être ainsi mieux distinguer les « raisonnements » de la machine et améliorer le code en conséquence si on voit des comportements peu optimisés ou bloquant l'accès à des meilleures solutions par exemple. Cela pourrait apporter une perspective nouvelle et pourrait favoriser l'invention de nouvelle stratégies.

Pour finir, je pense que l'exécution de mon code source est capable de trouver de très bons résultats sur les instances de moins de 150 villes et cela dans des temps raisonnables. J'admets cependant qu'il manque un peu d'efficacité pour réussir à produire des résultats d'aussi bonne qualité en des temps raisonnables sur des instances plus imposantes.